**Intro: what is {tidymodels}**

{tidymodels} is a meta-package has evolved quite a lot. If you don’t know what  
{tidymodels} is, it is a suite of packages that make machine learning with R a breeze. R has many  
packages for machine learning, each with their own syntax and function arguments. {tidymodels} aims  
at providing an unified interface which allows data scientists to focus on the problem they’re trying  
to solve, instead of wasting time with learning package specificities.

The packages included in {tidymodels} are:

* [{parsnip}](https://tidymodels.github.io/parsnip/articles/parsnip_Intro.html) for model definition
* [{recipes}](https://tidymodels.github.io/recipes/) for data preprocessing and feature engineering
* [{rsample}](https://tidymodels.github.io/rsample/) to resample data (useful for cross-validation)
* [{yardstick}](https://tidymodels.github.io/yardstick/index.html) to evaluate model performance
* [{dials}](https://tidymodels.github.io/dials/index.html) to define tuning parameters of your models
* [{tune}](https://tidymodels.github.io/tune/) for model tuning
* [{workflows}](https://tidymodels.github.io/workflows/) which allows you to bundle everything together and train models easily

There are some others, but I will not cover these. This is a lot of packages, and you might be  
worried of getting lost; however, in practice I noticed that loading {tidymodels} and then using  
the functions I needed was good enough. Only rarely did I need to know from which package a certain  
function came, and the more you use these, the better you know them, obviously. Before continuing,  
one final and important note: these packages are still in heavy development, so you might not want  
to use them in production yet. I don’t know how likely it is that the api still evolves, but my guess  
is that it is likely. However, even though it might be a bit early to use these packages for production  
code, I think it is important to learn about them as soon as possible and see what is possible with them.

As I will show you, these packages do make the process of training machine learning models a breeze, and of  
course they integrate very well with the rest of the {tidyverse} packages. The problem we’re going  
to tackle is to understand which variables play an important role in the probability of someone looking  
for a job. I’ll use Eustat’s microdata, and is called *Population with relation to activity (PRA)*.

**The problem at hand**

The dataset contains information on residents from the Basque country, and focuses on their labour supply.  
Thus, we have information on how many hours people work a week, if they work, in which industry, what  
is their educational attainment and whether they’re looking for a job.  
The first step, as usual, is to load the data and required packages:

library(tidyverse)

library(tidymodels)

library(readxl)

library(naniar)

library(janitor)

library(furrr)

list\_data <- Sys.glob("~/Documents/[b-rodrigues.github.com/content/blog/MICRO\*.csv](http://b-rodrigues.github.com/content/blog/MICRO*.csv)")

dataset <- map(list\_data, read\_csv2) %>%

bind\_rows()

dictionary <- read\_xlsx("~/Documents/b-rodrigues.github.com/content/blog/Microdatos\_PRA\_2019/diseño\_registro\_microdatos\_pra.xlsx", sheet="Valores",

col\_names = FALSE)

col\_names <- dictionary %>%

filter(![is.na](http://is.na)(...1)) %>%

dplyr::select(1:2)

english <- readRDS("~/Documents/[b-rodrigues.github.com/content/blog/english\_col\_names.rds](http://b-rodrigues.github.com/content/blog/english_col_names.rds)")

col\_names$english <- english

colnames(dataset) <- col\_names$english

dataset <- janitor::clean\_names(dataset)

Let’s take a look at the data:

head(dataset)

## # A tibble: 6 x 33

## household\_number survey\_year reference\_quart… territory capital sex

##

## 1 1 2019 1 48 9 6

## 2 1 2019 1 48 9 1

## 3 2 2019 1 48 1 1

## 4 2 2019 1 48 1 6

## 5 2 2019 1 48 1 6

## 6 2 2019 1 48 1 1

## # … with 27 more variables: place\_of\_birth , age , nationality ,

## # level\_of\_studies\_completed , ruled\_teaching\_system ,

## # occupational\_training , retirement\_situation ,

## # homework\_situation , part\_time\_employment ,

## # short\_time\_cause , job\_search , search\_reasons ,

## # day\_searched , make\_arrangements , search\_form ,

## # search\_months , availability ,

## # relationship\_with\_the\_activity ,

## # relationship\_with\_the\_activity\_2 , main\_occupation ,

## # main\_activity , main\_professional\_situation ,

## # main\_institutional\_sector , type\_of\_contract , hours ,

## # relationship , elevator

There are many columns, most of them are categorical variables and unfortunately the levels in the  
data are only some non-explicit codes. The excel file I have loaded, which I called dictionary  
contains the codes and their explanation. I kept the file opened while I was working, especially for  
missing values imputation. Indeed, there are missing values in the data, and one should always try  
to understand why before blindly imputing them. Indeed, there might be a very good reason why data  
might be missing for a particular column. For instance, if children are also surveyed, they would  
have an NA in the, say, main\_occupation column which gives the main occupation of the surveyed  
person. This might seem very obvious, but sometimes these reasons are not so obvious at all. You should  
always go back with such questions to the data owners/producers, because if not, you will certainly  
miss something very important. Anyway, the way I tackled this issue was by looking at the variables  
with missing data and checking two-way tables with other variables. For instance, to go back to my  
example from before, I would take a look at the two-way frequency table between age and main\_occupation.  
If all the missing values from main\_occupation where only for people 16 or younger, then it would  
be quite safe to assume that I was right, and I could recode these NAs in main\_occupation to  
"without occupation" for instance. I’ll spare you all this exploration, and go straight to the  
data cleaning:

dataset <- dataset %>%

mutate(main\_occupation2 = ifelse([is.na](http://is.na)(main\_occupation),

"without\_occupation",

main\_occupation))

dataset <- dataset %>%

mutate(main\_professional\_situation2 = ifelse([is.na](http://is.na)(main\_professional\_situation),

"without\_occupation",

main\_professional\_situation))

# People with missing hours are actually not working, so I put them to 0

dataset <- dataset %>%

mutate(hours = ifelse([is.na](http://is.na)(hours), 0, hours))

# Short time gives the reason why people are working less hours than specified in their contract

dataset <- dataset %>%

mutate(short\_time\_cause = ifelse(hours == 0 | [is.na](http://is.na)(short\_time\_cause),

"without\_occupation",

short\_time\_cause))

dataset <- dataset %>%

mutate(type\_of\_contract = ifelse([is.na](http://is.na)(type\_of\_contract),

"other\_contract",

type\_of\_contract))

Let’s now apply some further cleaning:

pra <- dataset %>%

filter(age %in% c("04", "05", "06", "07", "08", "09", "10", "11", "12", "13")) %>%

filter(retirement\_situation == 4) %>%

filter(![is.na](http://is.na)(job\_search)) %>%

select(capital, sex, place\_of\_birth, age, nationality, level\_of\_studies\_completed,

occupational\_training, job\_search, main\_occupation2, type\_of\_contract,

hours, short\_time\_cause, homework\_situation,

main\_professional\_situation2) %>%

mutate\_at(.vars = vars(-hours), .funs=as.character) %>%

mutate(job\_search = as.factor(job\_search))

I only keep people that are not retired and of ages where they could work. I remove rows where  
job\_search, the target, is missing, mutate all variables but hours to character and job\_search to factor. At  
first, I made every categorical column a factor but I got problems for certain models. I think the  
issue came from the recipe that I defined (I’ll talk about it below), but the problem was resolved  
if categorical variables were defined as character variables. However, for certain models, the target  
(I think it was xgboost) needs to be a factor variable for classification problems.

Let’s take a look at the data and check if any more data is missing:

str(pra)

## Classes 'spec\_tbl\_df', 'tbl\_df', 'tbl' and 'data.frame': 29083 obs. of 14 variables:

## $ capital : chr "9" "9" "1" "1" ...

## $ sex : chr "6" "1" "1" "6" ...

## $ place\_of\_birth : chr "1" "1" "1" "1" ...

## $ age : chr "09" "09" "11" "10" ...

## $ nationality : chr "1" "1" "1" "1" ...

## $ level\_of\_studies\_completed : chr "1" "2" "3" "3" ...

## $ occupational\_training : chr "N" "N" "N" "N" ...

## $ job\_search : Factor w/ 2 levels "N","S": 1 1 1 1 1 1 1 1 1 1 ...

## $ main\_occupation2 : chr "5" "7" "3" "2" ...

## $ type\_of\_contract : chr "1" "other\_contract" "other\_contract" "1" ...

## $ hours : num 36 40 40 40 0 0 22 38 40 0 ...

## $ short\_time\_cause : chr "2" "2" "2" "2" ...

## $ homework\_situation : chr "1" "2" "2" "2" ...

## $ main\_professional\_situation2: chr "4" "2" "3" "4" ...

vis\_miss(pra)

The final dataset contains 29083 observations. Look’s like we’re good to go.

**Setting up the training: resampling**

In order to properly train a model, one needs to split the data into two: a part for trying out  
models with different configuration of hyper-parameters, and another part for final evaluation of  
the model. This is achieved with rsample::initial\_split():

pra\_split <- initial\_split(pra, prop = 0.9)

pra\_split now contains a training set and a testing set. We can get these by using the  
rsample::training() and rsample::testing() functions:

pra\_train <- training(pra\_split)

pra\_test <- testing(pra\_split)

We can’t stop here though. First we need to split the training set further, in order to perform  
cross validation. Cross validation will allow us to select the best model; by best I mean a model  
that has a good hyper-parameter configuration, enabling the model to generalize well to unseen data.  
I do this by creating 10 splits from the training data (I won’t touch the testing data up until  
the very end. This testing data is thus sometimes called the holdout set as well):

pra\_cv\_splits <- vfold\_cv(pra\_train, v = 10)

Let’s take a look at this object:

pra\_cv\_splits

## # 10-fold cross-validation

## # A tibble: 10 x 2

## splits id

##

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

**Preprocessing the data**

I have already pre-processed the missing values in the dataset, so there is not much more that  
I can do. I will simply create dummy variables out of the categorical variables using step\_dummy():

preprocess <- recipe(job\_search ~ ., data = pra) %>%

step\_dummy(all\_predictors())

preprocess is a recipe that defines the transformations that must be applied to the training data  
before fitting. In this case there is only one step; transforming all the predictors into dummies  
(hours is a numeric variable and will be ignored by this step). The recipe also defines the  
formula that will be fitted by the models, job\_search ~ ., and takes data as a further argument.  
This is only to give the data frame specification to recipe(): it could even be an empty data frame  
with the right column names and types. This is why I give it the original data pra and not the  
training set pra\_train. Because this recipe is very simple, it could be applied to the original  
raw data pra and then I could do the split into training and testing set, as well as further  
splitting the training set into 10 cross-validation sets. However, this is not the recommended way  
of applying pre-processing steps. Pre-processing needs to happen inside the cross-validation loop,  
not outside of it. Why? Suppose that you are normalizing a numeric variable, meaning, substracting  
its mean from it and dividing by its standard deviation. If you do this operation outside of  
cross-validation, and even worse, before splitting the data into training and testing set, you will  
be leaking information from the testing set into the training set. The mean will contain information  
from the testing set, which will be picked up by the model.  
It is much better and “realistic” to first split the data and then apply  
the pre-processing (remember that *hiding* the test set from the model is supposed to simulate  
the fact that new, completely unseen data, is thrown at your model once it’s put into production). The  
same logic applies to cross-validation splits; each split contains now also a training and a testing  
set (which I will be calling analysis and assessment sets, following {tidymodels}’s author,  
[Max Kuhn](https://twitter.com/topepos/status/1066131042615140353?s=20)) and thus the pre-processing needs to be applied inside the cross-validation loop, meaning that the analysis set will be processed on the fly.

**Model definition**

We come now to the very interesting part: model definition. With {parsnip}, another {tidymodels}  
package, defining models is always the same, regardless of the underlying package doing the heavy  
lifting. For instance, to define a logistic regression one would simply write:

# logistic regression

logit\_tune\_pra <- logistic\_reg() %>%

set\_engine("glm")

This defines a standard logistic regression, powered by the glm() *engine* or function. The way  
to do this in vanilla R would be :

glm(y ~ ., data = mydata, family = "binomial")

The difference here is that the formula is contained in the glm() function; in our case it is  
contained in the recipe, which is why I don’t repeat it in the model definition above. You might  
wonder what the added value of using {tidymodels} for this is. Well, suppose now that I would like  
to run a logistic regression but with regularization. I would use {glmnet} for this but would need  
to know the specific syntax of glmnet() which, as you will see, is very different than the one  
for glm():

glmnet(x\_vars[train,], y\_var[train], alpha = 1, lambda = 1.6)

glmnet(), unlike glm(), does not use a formula as an input, but two matrices, one for the design  
matrix, and another for the target variable. Using {parsnip}, however, I simply need to change the  
engine from "glm" to "glmnet":

# logistic regression

logit\_tune\_pra <- logistic\_reg() %>%

set\_engine("glmnet")

This makes things much simpler as now users only need to learn how to use {parsnip}. However,  
it is of course still important to read the documentation of the original packages, because it is  
were hyper-parameters are discussed. Another advantage of {parsnip} is that the same words  
are used to speak of the same hyper-parameters . For instance for tree-based methods, the number of  
trees is sometimes ntree then in another package num\_trees, and is again different in yet another package.  
In {parsnip}’s interface for tree-based methods, this parameter is simply  
called tree. Users can fix the value of hyper-parameters directly by passing values to, say, tree  
(as in "tree" = 200), or they can tune these hyper-parameters. To do so, one needs to tag them, like so:

# logistic regression

logit\_tune\_pra <- logistic\_reg(penalty = tune(), mixture = tune()) %>%

set\_engine("glmnet")

This defines logit\_tune\_pra with 2 hyper-parameters that must be tuned using cross-validation,  
the penalty and the amount of mixture between penalties (this is for elasticnet regularization).

Now, I will define 5 different models, with different hyper-parameters to tune, and I will also  
define a grid of hyper-parameters of size 10 for each model. This means that I will train these 5  
models 10 times, each time with a different hyper-parameter configuration. To define the grid, I use  
the grid\_max\_entropy() function from the {dials} package. This creates a grid with points that  
are randomly drawn from the parameter space in a way that ensures that the combination we get  
covers the whole space, or at least are not too far away from any portion of the space. Of course,  
the more configuration you try, the better, but the longer the training will run.

# Logistic regression

logit\_tune\_pra <- logistic\_reg(penalty = tune(), mixture = tune()) %>%

set\_engine("glmnet")

# Hyperparameter grid

logit\_grid <- logit\_tune\_pra %>%

parameters() %>%

grid\_max\_entropy(size = 10)

# Workflow bundling every step

logit\_wflow <- workflow() %>%

add\_recipe(preprocess) %>%

add\_model(logit\_tune\_pra)

# random forest

rf\_tune\_pra <- rand\_forest(mtry = tune(), trees = tune()) %>%

set\_engine("ranger") %>%

set\_mode("classification")

rf\_grid <- rf\_tune\_pra %>%

parameters() %>%

finalize(select(pra, -job\_search)) %>%

grid\_max\_entropy(size = 10)

rf\_wflow <- workflow() %>%

add\_recipe(preprocess) %>%

add\_model(rf\_tune\_pra)

# mars model

mars\_tune\_pra <- mars(num\_terms = tune(), prod\_degree = 2, prune\_method = tune()) %>%

set\_engine("earth") %>%

set\_mode("classification")

mars\_grid <- mars\_tune\_pra %>%

parameters() %>%

grid\_max\_entropy(size = 10)

mars\_wflow <- workflow() %>%

add\_recipe(preprocess) %>%

add\_model(mars\_tune\_pra)

#boosted trees

boost\_tune\_pra <- boost\_tree(mtry = tune(), tree = tune(),

learn\_rate = tune(), tree\_depth = tune()) %>%

set\_engine("xgboost") %>%

set\_mode("classification")

boost\_grid <- boost\_tune\_pra %>%

parameters() %>%

finalize(select(pra, -job\_search)) %>%

grid\_max\_entropy(size = 10)

boost\_wflow <- workflow() %>%

add\_recipe(preprocess) %>%

add\_model(boost\_tune\_pra)

#neural nets

keras\_tune\_pra <- mlp(hidden\_units = tune(), penalty = tune(), activation = "relu") %>%

set\_engine("keras") %>%

set\_mode("classification")

keras\_grid <- keras\_tune\_pra %>%

parameters() %>%

grid\_max\_entropy(size = 10)

keras\_wflow <- workflow() %>%

add\_recipe(preprocess) %>%

add\_model(keras\_tune\_pra)

For each model, I defined three objects; the model itself, for instance keras\_tune\_pra, then a  
grid of hyper-parameters, and finally a workflow. To define the grid, I need to extract the parameters  
to tune using the parameters() function, and for tree based methods, I also need to use finalize()  
to set the mtry parameter. This is because mtry depends on the dimensions of the data (the value  
of mtry cannot be larger than the number of features), so I need to pass on this information  
to…well, finalize the grid. Then I can choose the size of the grid and how I want to create it  
(randomly, or using max entropy, or regularly spaced…).  
A workflow bundles the pre-processing and the model definition together, and makes fitting the model  
very easy. Workflows make it easy to run the pre-processing inside the cross-validation loop.  
Workflow objects can be passed to the fitting function, as we shall see in the next section.

**Fitting models with {tidymodels}**

Fitting one model with {tidymodels} is quite easy:

fitted\_model <- fit(model\_formula, data = data\_train)

and that’s it. If you define a workflow, which bundles pre-processing and model definition  
in one package, you need to pass it to fit() as well:

fitted\_wflow <- fit(model\_wflow, data = data\_train)

However, a single call to fit does not perform cross-validation. This simply trains the model on  
the training data, and that’s it. To perform cross validation, you can use either fit\_resamples():

fitted\_resamples <- fit\_resamples(model\_wflow,

resamples = my\_cv\_splits,

control = control\_resamples(save\_pred = TRUE))

or tune\_grid():

tuned\_model <- tune\_grid(model\_wflow,

resamples = my\_cv\_splits,

grid = my\_grid,

control = control\_resamples(save\_pred = TRUE))

As you probably guessed it, fit\_resamples() does not perform tuning; it simply fits a model  
specification (without varying hyper-parameters) to all the analysis sets contained in the  
my\_cv\_splits object (which contains the resampled training data for cross-validation), while  
tune\_grid() does the same, but allows for varying hyper-parameters.

We thus are going to use tune\_grid() to fit our models and perform hyper-paramater tuning.  
However, since I have 5 models and 5 grids, I’ll be using map2() for this. If you’re not familiar  
with map2(), here’s a quick example:

map2(c(1, 1, 1), c(2,2,2), `+`)

## [[1]]

## [1] 3

##

## [[2]]

## [1] 3

##

## [[3]]

## [1] 3

map2() maps the +() function to each element of both vectors successively. I’m going to use  
this to map the tune\_grid() function to a list of models and a list of grids. But because this is  
going to take some time to run, and because I have an AMD Ryzen 5 1600X processor with 6 physical  
cores and 12 logical cores, I’ll by running this in parallel using furrr::future\_map2().

furrr::future\_map2() will run one model per core, and the way to do it is to simply define  
how many cores I want to use, then replace map2() in my code by future\_map2():

wflow\_list <- list(logit\_wflow, rf\_wflow, mars\_wflow, boost\_wflow, keras\_wflow)

grid\_list <- list(logit\_grid, rf\_grid, mars\_grid, boost\_grid, keras\_grid)

plan(multiprocess, workers = 6)

trained\_models\_list <- future\_map2(.x = wflow\_list,

.y = grid\_list,

~tune\_grid(.x , resamples = pra\_cv\_splits, grid = .y))

Running this code took almost 3 hours. In the end, here is the result:

trained\_models\_list

## [[1]]

## # 10-fold cross-validation

## # A tibble: 10 x 4

## splits id .metrics .notes

## \*

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

##

## [[2]]

## # 10-fold cross-validation

## # A tibble: 10 x 4

## splits id .metrics .notes

## \*

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

##

## [[3]]

## # 10-fold cross-validation

## # A tibble: 10 x 4

## splits id .metrics .notes

## \*

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

##

## [[4]]

## # 10-fold cross-validation

## # A tibble: 10 x 4

## splits id .metrics .notes

## \*

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

##

## [[5]]

## # 10-fold cross-validation

## # A tibble: 10 x 4

## splits id .metrics .notes

## \*

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

I now have a list of 5 tibbles containing the analysis/assessment splits, the id identifying the  
cross-validation fold, a list-column containing information on model performance for that given  
split and some notes (if everything goes well, notes are empty). Let’s take a look at the column  
.metrics of the first model and for the first fold:

trained\_models\_list[[1]]$.metrics[[1]]

## # A tibble: 20 x 5

## penalty mixture .metric .estimator .estimate

##

## 1 4.25e- 3 0.0615 accuracy binary 0.906

## 2 4.25e- 3 0.0615 roc\_auc binary 0.895

## 3 6.57e-10 0.0655 accuracy binary 0.908

## 4 6.57e-10 0.0655 roc\_auc binary 0.897

## 5 1.18e- 6 0.167 accuracy binary 0.908

## 6 1.18e- 6 0.167 roc\_auc binary 0.897

## 7 2.19e-10 0.371 accuracy binary 0.907

## 8 2.19e-10 0.371 roc\_auc binary 0.897

## 9 2.73e- 1 0.397 accuracy binary 0.885

## 10 2.73e- 1 0.397 roc\_auc binary 0.5

## 11 1.72e- 6 0.504 accuracy binary 0.907

## 12 1.72e- 6 0.504 roc\_auc binary 0.897

## 13 1.25e- 9 0.633 accuracy binary 0.907

## 14 1.25e- 9 0.633 roc\_auc binary 0.897

## 15 6.62e- 6 0.880 accuracy binary 0.907

## 16 6.62e- 6 0.880 roc\_auc binary 0.897

## 17 6.00e- 1 0.899 accuracy binary 0.885

## 18 6.00e- 1 0.899 roc\_auc binary 0.5

## 19 4.57e-10 0.989 accuracy binary 0.907

## 20 4.57e-10 0.989 roc\_auc binary 0.897

This shows how the 10 different configurations of the elasticnet model performed. To see how the  
model performed on the second fold:

trained\_models\_list[[1]]$.metrics[[2]]

## # A tibble: 20 x 5

## penalty mixture .metric .estimator .estimate

##

## 1 4.25e- 3 0.0615 accuracy binary 0.913

## 2 4.25e- 3 0.0615 roc\_auc binary 0.874

## 3 6.57e-10 0.0655 accuracy binary 0.913

## 4 6.57e-10 0.0655 roc\_auc binary 0.877

## 5 1.18e- 6 0.167 accuracy binary 0.913

## 6 1.18e- 6 0.167 roc\_auc binary 0.878

## 7 2.19e-10 0.371 accuracy binary 0.913

## 8 2.19e-10 0.371 roc\_auc binary 0.878

## 9 2.73e- 1 0.397 accuracy binary 0.901

## 10 2.73e- 1 0.397 roc\_auc binary 0.5

## 11 1.72e- 6 0.504 accuracy binary 0.913

## 12 1.72e- 6 0.504 roc\_auc binary 0.878

## 13 1.25e- 9 0.633 accuracy binary 0.913

## 14 1.25e- 9 0.633 roc\_auc binary 0.878

## 15 6.62e- 6 0.880 accuracy binary 0.913

## 16 6.62e- 6 0.880 roc\_auc binary 0.878

## 17 6.00e- 1 0.899 accuracy binary 0.901

## 18 6.00e- 1 0.899 roc\_auc binary 0.5

## 19 4.57e-10 0.989 accuracy binary 0.913

## 20 4.57e-10 0.989 roc\_auc binary 0.878

Hyper-Parameters are the same; it is only the cross validation fold that is different. To get the  
best performing model from such objects you can use show\_best() which will extract the best  
performing models across all the cross validation folds:

show\_best(trained\_models\_list[[1]], metric = "accuracy")

## # A tibble: 5 x 7

## penalty mixture .metric .estimator mean n std\_err

##

## 1 6.57e-10 0.0655 accuracy binary 0.916 10 0.00179

## 2 1.18e- 6 0.167 accuracy binary 0.916 10 0.00180

## 3 1.72e- 6 0.504 accuracy binary 0.916 10 0.00182

## 4 4.57e-10 0.989 accuracy binary 0.916 10 0.00181

## 5 6.62e- 6 0.880 accuracy binary 0.916 10 0.00181

This shows the 5 best configurations for elasticnet when looking at accuracy. Now how to get the best  
performing elasticnet regression, random forest, boosted trees, etc? Easy, using map():

map(trained\_models\_list, show\_best, metric = "accuracy")

## [[1]]

## # A tibble: 5 x 7

## penalty mixture .metric .estimator mean n std\_err

##

## 1 6.57e-10 0.0655 accuracy binary 0.916 10 0.00179

## 2 1.18e- 6 0.167 accuracy binary 0.916 10 0.00180

## 3 1.72e- 6 0.504 accuracy binary 0.916 10 0.00182

## 4 4.57e-10 0.989 accuracy binary 0.916 10 0.00181

## 5 6.62e- 6 0.880 accuracy binary 0.916 10 0.00181

##

## [[2]]

## # A tibble: 5 x 7

## mtry trees .metric .estimator mean n std\_err

##

## 1 13 1991 accuracy binary 0.929 10 0.00172

## 2 13 1180 accuracy binary 0.929 10 0.00168

## 3 12 285 accuracy binary 0.928 10 0.00168

## 4 8 1567 accuracy binary 0.927 10 0.00171

## 5 8 647 accuracy binary 0.927 10 0.00191

##

## [[3]]

## # A tibble: 5 x 7

## num\_terms prune\_method .metric .estimator mean n std\_err

##

## 1 5 backward accuracy binary 0.904 10 0.00186

## 2 5 forward accuracy binary 0.902 10 0.00185

## 3 4 exhaustive accuracy binary 0.901 10 0.00167

## 4 4 seqrep accuracy binary 0.901 10 0.00167

## 5 2 backward accuracy binary 0.896 10 0.00209

##

## [[4]]

## # A tibble: 5 x 9

## mtry trees tree\_depth learn\_rate .metric .estimator mean n std\_err

##

## 1 12 1245 12 7.70e- 2 accuracy binary 0.929 10 0.00175

## 2 1 239 8 8.23e- 2 accuracy binary 0.927 10 0.00186

## 3 1 835 14 8.53e-10 accuracy binary 0.913 10 0.00232

## 4 4 1522 12 2.22e- 5 accuracy binary 0.896 10 0.00209

## 5 6 313 2 1.21e- 8 accuracy binary 0.896 10 0.00209

##

## [[5]]

## # A tibble: 5 x 7

## hidden\_units penalty .metric .estimator mean n std\_err

##

## 1 10 3.07e- 6 accuracy binary 0.917 10 0.00209

## 2 6 1.69e-10 accuracy binary 0.917 10 0.00216

## 3 4 2.32e- 7 accuracy binary 0.916 10 0.00194

## 4 7 5.52e- 5 accuracy binary 0.916 10 0.00163

## 5 8 1.13e- 9 accuracy binary 0.916 10 0.00173

Now, we need to test these models on the holdout set, but this post is already quite long. In the next  
blog post, I will retrain the top best performing models for each type of model and see how they  
fare against the holdout set. I’ll be also looking at explainability, so stay tuned!